International Journal of Software Engineering & Knowledge Engineering
Vol. 11, No. 2 (2001) 139–156
© World Scientific Publishing Company

A CONSTRAINT SOLVER AND ITS APPLICATION TO PATH FEASIBILITY ANALYSIS*

JIAN ZHANG and XIAOXU WANG
Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing 100080, China

Many testing methods are based on program paths. A well-known problem with them is that some paths are infeasible. To decide the feasibility of paths, we may solve a set of constraints. In this paper, we describe constraint-based tools that can be used for this purpose. They accept constraints expressed in a natural form, which may involve variables of different types such as integers, Booleans, reals and fixed-size arrays. The constraint solver is an extension of a Boolean satisfiability checker and it makes use of a linear programming package. The solving algorithm is described, and examples are given to illustrate the use of the tools. For many paths in the testing literature, their feasibility can be decided in a reasonable amount of time.

Keywords: Constraint solving; test data generation; infeasible paths; arrays.

1. Introduction

Program testing methods can be roughly divided into two categories: black-box and white-box. The former is based on the functional specification, while the latter uses the source code of the program.

Many white-box testing methods can be viewed as *path-oriented*. We select a set of paths of the program, such that some criteria are met. A commonly used criterion is *statement coverage*, which means that all statements should be included in the chosen set of paths. There are also other criteria such as branch coverage and path coverage.

To execute each path, we need to find appropriate values for the input variables. This problem is called *test data generation*. If no values of the variables can cause the program to be executed along a path, we say that the path is *infeasible* or *non-executable*. A well-known problem in program testing is that many paths are infeasible [1, 2]. In this paper, we shall describe constraint-based tools and their application to the analysis of path feasibility.

*A preliminary version of this paper was presented at the *First Asia-Pacific Conference on Quality Software* (APAQS 2000) [13]. This work was supported by the Natural Science Foundation of China (NSFC) under grant no. 69703014 and the National Key Basic Research Plan (NKBRSF) under grant no. G1998030600.

This paper is organized as follows. In the next section, we recall some basic concepts of path-oriented testing and analysis. Then in Sec. 3, we briefly show how to derive a set of constraints from a path automatically. The path is executable if these constraints are satisfiable. In Sec. 4, we describe a constraint solver and its use in the analysis of path feasibility. Finally, our work is compared with other related approaches, and some further research topics are discussed.

2. Path-based Analysis and Testing

The structure of a program can often be described by its *control flow graph* (CFG), which is a directed graph. In a CFG, a node corresponds to a simple statement or the evaluation of a logical expression in a compound statement. The edges in the graph denote potential flow of control between the statements. A control flow path of the program is represented by a path in a CFG, which starts from the entry node. It can be denoted by a sequence of nodes, or more explicitly, by a sequence of logical expressions and simple statements.

For simplicity, we do not consider subroutines or function calls within the program body. A "program" is identified with a procedure. We shall use a C-like syntax to describe programs and algorithms. The symbol '`=`' means assignment, while '`==`' means equality. The logical operators NOT, OR, AND are denoted by '`!`', '`||`', '`&&`', respectively. An array `a` of size `N` consists of the elements `a[0]`, `a[1]`, `...`, `a[N-1]`. Many example programs in the literature adopt the syntax of Pascal. To be consistent, we may add an element (i.e., `a[0]`) to the array, without using it in the program. Then the body of the original example does not need to be changed.

Example 1. The following procedure computes the quotient and remainder of two positive integers.

```
void qr(int m, int n)
{
          int q, r;
  /*S1*/  r = m;
  /*S2*/  q = 0;
  /*S3*/  while (r >= n) {
  /*S4*/    r = r-n;
  /*S5*/    q = q+1;
          }
  /*S6*/  printf(" %d %d\n",q,r);
}
```

Here m and n are the input integers, q is the quotient, and r is the remainder. S1, . . . , S6 are labels of the statements. The CFG is given in Fig. 1.

The following are two paths of the program:

Path 1: S1 S2 S3 S6.

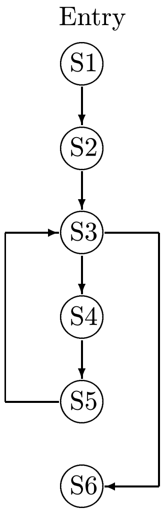


Fig. 1. Control flow graph for the program *qr*.

Path 2: S1 S2 S3 S4 S5 S3 S6.

The first path may also be represented by the sequence:

```
r = m;  q = 0;  !(r >= n);  printf(" %d %d\n",q,r);
```

For the sake of brevity, we assume that a path is a sequence of logical expressions and assignment statements. Output statements like `printf` may be omitted or replaced by the Boolean constant TRUE, because normally they do not affect the program's flow of control. If the path contains input statements, we can introduce auxiliary variables `$inputVar_i (i = 1, 2, ...)`. Then the statement `scanf("%d",&n)` can be replaced by the assignment `n = $inputVar_1`.

The execution of a (deterministic) program depends on the initial values of the input variables. Let us consider the above example. If the procedure is called with the parameters $m = 3, n = 2$, the procedure *qr* will be executed along Path 2. On the other hand, if $m = 2, n = 3$ initially, *qr* will be executed along Path 1.

Sometimes, programmers use assertions to increase the reliability of programs. The assertions can be checked at run-time to detect errors. Besides that, one can also combine test data generation techniques with assertions, so as to detect more bugs. Assertion-oriented testing is discussed in [3]. There, the assertions are translated into procedural code, and then an execution-based testing method is applied to the new code.

For a simple Boolean assertion, we can insert its *negation* into a path and analyze the feasibility of the new path *statically*. An error exists, if there are input data which cause the program to be executed up to the point of an assertion, and the negated assertion is satisfied.

For the quotient/remainder routine in Example 1, usually we can write the postcondition $0 \le r < n$. We may insert its negation to Path 1, and consider the new path:

```
r = m;  q = 0;  !(r >= n);  (r < 0) || (r >= n);
```

It is executable if the value of m is negative initially. But as we mentioned, the parameters m and n should be positive integers. If these preconditions are added, we obtain an infeasible path:

```
m > 0; n > 0;  r = m;  q = 0;  !(r >= n);  (r < 0)||(r >= n);
```

The infeasibility implies that the postcondition holds when the program executes along Path 1.

Syntactically, the negated assertions are similar to other logical expressions.

In summary, for the purpose of testing or static analysis, we can select a set of paths from the control flow graph, and then try to find input data to execute them. Various methods for path selection have been proposed [4]. In this paper, we are mainly concerned with algorithms for finding appropriate input data such that the program is executed along a given path.

3. Extracting Path Conditions

For a general class of programs, many analysis problems are undecidable [5]. For instance, there is no algorithm that can tell us whether an arbitrary statement is reachable. Nor is there an algorithm for deciding the feasibility of program paths.

However, if some restrictions are put on the programs, the problems may become decidable. In the present work, we assume that numeric expressions (involving integer and real variables) are *linear*. It is believed that, in many production software systems, a large percentage of expressions and predicates are linear [6].

Similar to some existing works [7, 8], we generate test data in two steps: firstly extract a set of constraints from the given path, and then solve the constraints.

Given a path, we can obtain a set of constraints called *path predicates* or *path conditions.* The path is executable if and only if these constraints are satisfiable.

Basically, there are two different ways of extracting the path conditions [7]:

- *Forward expansion* starts from the entry node, and builds symbolic expressions as each statement in the path is interpreted.
- *Backward substitution* starts from the final node and proceeds to the entry node, while keeping a set of constraints on the input variables.

In this paper, we focus on the backward substitution method.

Given a path P, let PC_P denote the set of path conditions. As mentioned previously, P is a sequence of logical expressions and assignment statements. We use $Elem_P[i]$ to denote the i'th element of the sequence, and use Len_P to denote P's length (i.e., the number of logical expressions and assignments in P). The subscript

P may be omitted when no confusion occurs. Then PC can be obtained in the following way:

```
for($i = Len - 1$; $i \geq 0$; $i - -$)
    if ($Elem[i]$ is a logical expression)
        $PC = PC \cup \{ Elem[i] \}$;
    else /* $Elem[i]$ is an assignment */
        substitute($PC$, $Elem[i]$);
```

The procedure substitute() changes the PC using an assignment. Firstly we consider the trivial case when the left-hand side of the assignment is a simple variable x. In this case, every occurrence of x in PC is replaced by the right-hand side of the assignment.

As a simple example, let us consider the following path:

```
n > 9;  n = n+1;  n < 3;
```

At first, we have the expression `n < 3`, which is added to the PC. Then comes the assignment `n = n+1`. So the constraint in the PC is changed to `n+1 < 3`. Finally, the expression `n > 9` is added, and inconsistency occurs. Thus the path is infeasible.

We have implemented an automated tool for path analysis and testing, called PAT. The input consists of variable declarations followed by a path (i.e., a sequence of assignments and logical expressions). The output is a set of constraints (i.e., the path conditions). During the substitution, we make some simplifications to the constraints so that they do not get too complicated. For instance, the constraint $i + 5 > j - 1$ is transformed into the simpler one: $i + 6 - j > 0$. The constraint TRUE is always omitted. As soon as the constraint FALSE is generated, we know that the path is infeasible, and the constraint generation process stops.

Example 1. (Cont'd) Suppose we need input data such that the procedure *qr* executes along Path 2. We may specify the path as follows:

```
   int m, n, q, r;
{
     r = m;
     q = 0;
   @ r >= n;
     r = r-n;
     q = q+1;
   @ r < n;
}
```

Here we use the symbol '@' to distinguish logical expressions from assignment statements. Our tool PAT generates the following constraints:

```
m-n >= 0;  m-2n < 0;
```

The program will be executed along Path 2, if initially the positive integers m and n satisfy the above two constraints.

Now we discuss the behavior of the procedure `substitute()` in the presence of array assignments. Arrays are often used in most programs. Without loss of generality, we consider one-dimensional arrays only. We also assume that there are no nested array expressions like $a[b[i]]$. During backward substitution, if the left-hand side of the assignment is an array variable $a[e_0]$, we look for all sub-expressions $a[e_i]$ $(1 \leq i \leq m)$ in the current PC. Here each e_j $(0 \leq j \leq m)$ is a numeric expression involving simple variables. From the PC, we generate 2^m new cases, each of which corresponds to a combination of equations or inequalities between e_0 and e_i $(1 \leq i \leq m)$.

For example, suppose the current constraint is $a[i]+a[j] > 3$, and the assignment is $a[k] = c$. Then the new constraint will be the following:

```
   ((k == i) && (k == j) && (2c > 3))
|| ((k == i) && (k != j) && (c+a[j] > 3))
|| ((k != i) && (k == j) && (a[i]+c > 3))
|| ((k != i) && (k != j) && (a[i]+a[j] > 3))
```

The above shows the most general situation. It is possible that some disjunct can be eliminated. For example, if $i = 1$, $j = 2$, then k cannot be equal to both i and j. So the first line of the new constraint is not necessary.

4. Constraint Solving

After the path conditions are obtained, we need to decide whether they are satisfiable or not. For satisfiable conditions, we often have to find the variables' values to satisfy them. This kind of problems have been studied by many researchers in the artificial intelligence and operations research communities.

Informally speaking, a *Constraint Satisfaction Problem* (CSP) [9] consists of a set of variables, each of which can take values from some domain. In addition, there are some constraints defined on the variables. Solving a CSP means finding a value for each variable, such that all the constraints hold.

Obviously, CSP represents a very general class of problems. In practice, one tends to be more interested in specific forms of constraints.

4.1. *Linear programming*

In Example 1, the constraints generated by the tool PAT are linear inequalities defined on integer variables. This kind of constraints often occur in various applications.

A *linear programming* (LP) problem [10] has a set of variables, a linear function of the variables (called the *objective function*) and a set of linear inequalities or equations (called *constraints*). The goal is to find values of the variables which satisfy the constraints, such that the objective function has the maximum or minimum value. Usually the variables can take real numbers as their values. If the values of some variables are restricted to be integers, the problem is known as mixed integer (linear) programming [10].

The following is a simple example of LP:

$$\begin{array}{l} \text{Maximize} \quad 3y - x \\ \text{subject to the constraints} \\ \quad x + 5y \leq 8.4, \\ \quad 2x + y \geq 6.9. \end{array}$$

Its solution is as follows. When $x = 2.9$, $y = 1.1$, the objective function $(3y - x)$ achieves the maximum value 0.4.

In the normal case, the problem has a solution which satisfies the constraints and maximizes (or minimizes) the objective function. But some problems may be *infeasible* or *unbounded*. The former means that the constraints are contradictory, while the latter means that the objective function does not have a finite bound.

LP is tractable, and there are many efficient software packages for solving such problems. Some of them are commercial products, like LINDO (available at `http://www.lindo.com`). But there are also some public domain software, like `lp_solve`. It is a mixed integer linear program solver, developed by Michel Berkelaar. It is available from `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`.
To solve the previous example, one may give the following as input to `lp_solve`:

```
max: 3y - x;
x + 5y <= 8.4;
2x + y >= 6.9;
```

Linear programming has been used to solve path conditions and to generate test cases [11]. However, LP software cannot be used directly when logical operators occur in the constraints, e.g.,

```
(x > 5 || y + 2z == 4) && (z <= 3 || !(x-y == 0))
```

4.2. *Boolean satisfiability*

A Boolean variable can only be assigned some truth value (TRUE or FALSE). A Boolean formula is constructed from a set of Boolean variables using the logical operators AND (i.e., conjunction), OR (i.e., disjunction), etc. A literal is a variable or its negation. The disjunction of a set of literals is a clause. A Boolean formula can be transformed into a conjunction of clauses.

If we can assign a truth value to each variable such that the whole formula evaluates to TRUE, then the formula is said to be satisfiable. Deciding the satisfiability of Boolean formulas can be regarded as a special case of the CSP. If the formula is in conjunctive normal form (CNF), i.e., it is a conjunction of clauses, the problem is well-known as SAT. This is the first NP-hard problem [12].

Many algorithms and methods have been proposed to solve the satisfiability problem. Some of them are based on backtracking search, and some are based on local search. In this paper, we focus on backtracking algorithms. Such an algorithm works on *partial solutions*. In a partial solution, some variables are assigned values. Initially, no variable has a value. The algorithm can be represented by the recursive function in Fig. 2. The parameters $pSol$ and $Fmla$ denote a partial solution and the input Boolean formula, respectively.

```
Boolean BSat($pSol, Fmla$)
{
    BPropagate($pSol, Fmla$);
    if contradiction results
        return FALSE;
    if $pSol$ assigns a truth value to every Boolean variable
        return TRUE;
    choose a variable $x$ that does not have a value in $pSol$;
    if BSat($pSol \cup \{x$ = TRUE$\}$, $Fmla$)
        return TRUE;
    if BSat($pSol \cup \{x$ = FALSE$\}$, $Fmla$)
        return TRUE;
    return FALSE;
}
```

Fig. 2. Deciding the satisfiability of a Boolean formula.

In the algorithm, BPropagate() is a procedure for simplifying the formula $Fmla$ using the assignments of $pSol$. For example, if $pSol$ is `x = TRUE, y = FALSE` , and $Fmla$ is `(x || z) && (b || y)`, the formula will be simplified to `b`. On the other hand, if $Fmla$ is `(x && y)`, contradiction occurs.

The execution of a backtracking algorithm may be depicted as a search tree. Each branch of the tree corresponds to a partial solution. (See Example 2 below.)

4.3. *BoNuS*

In many cases, a constraint involves both Boolean operators and numerical expressions. We have implemented a constraint solver called BoNuS [13], which is an extension of a Boolean satisfiability checker. In a constraint satisfaction problem accepted by BoNuS, the variables can be Booleans, integers, reals or enumerated. Each constraint is a Boolean combination of primitive constraints, while a primitive constraint is a Boolean variable or a comparison between two numeric expressions.

Example 2. Given the following input:

```
  int  age, salary;
  enum gender: female, male;
  bool married;
  bool b1 = (age > 18);
  bool b2 = (gender == female);
  bool b3 = (salary > 10000);
  bool b4 = (salary <  8000);

{
  imp(married, b1);
  imp(or(married,b2), b4);
  and(b2, not(b1));
  or(b3,married);
}
```

BoNuS can decide automatically that the problem has no solution. The problem has two integer variables and one enumerated variable. In addition, there are 5 Boolean variables, 4 of which stand for numeric constraints. There are 4 Boolean formulas to be satisfied. The symbol `imp` stands for implication. So the second formula says, if a person is married or is female, the salary is less than 8000.

BoNuS is written using the parsing tool Yacc and the LP library `lp_solve` (mentioned previously). It can be regarded as an extension of a Boolean satisfiability checker. The modifications are mainly the following:

(I) Prior to the Search

For an enumerated variable v that can take values from the set $\{e_1, e_2, \ldots, e_n\}$, we introduce n new Boolean variables $\$ve_1$, $\$ve_2$, …, $\$ve_n$. The condition "$v = e_i$" is replaced by $\$ve_i$. To ensure soundness, we need to add two formulas indicating that the values are all inclusive and mutually exclusive:

$$\begin{array}{l}\$ve_1 \vee \$ve_2 \vee \ldots \vee \$ve_n ,\\ LTE1(\$ve_1, \$ve_2, \ldots, \$ve_n)\,.\end{array}$$

Here $LTE1$ is a Boolean operator which means, among the n arguments, at most one is true.

A primitive constraint can be an inequality or an equation. BoNuS has a preprocessing step which transforms equations into inequalities. This is done in the following way, which is straightforward. Suppose that there is a Boolean variable b_i that stands for the primitive constraint $Exp_1 = Exp_2$. Then we introduce two new Boolean variables:

$$b_{i_1} :\ Exp_1 \leq Exp_2, \qquad b_{i_2} :\ Exp_1 \geq Exp_2$$

and replace b_i by $and(b_{i_1}, b_{i_2})$. Moreover, we have to add the condition $or(b_{i_1}, b_{i_2})$ to the Boolean formula. Otherwise, we might get a solution in which b_{i_1} and b_{i_2} are both false. Disequations like $Exp_1 \neq Exp_2$ are also transformed in a similar way.

(II) During the Search

The search algorithm of BoNuS is essentially the same as the procedure BSat() (Fig. 2), except for the constraint propagation step. The procedure BPropagate() only uses inference rules from the Boolean logic (or the propositional logic). However, in BoNuS, numeric expressions have to be considered.

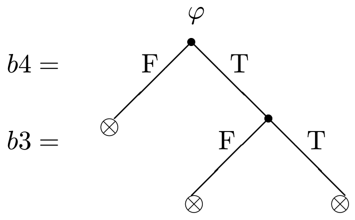
In the input of BoNuS, if a Boolean variable `nb` stands for a numeric constraint, we denote that primitive constraint by `prCons(nb)`. During the search process, when we got a partial solution *pSol* which assigns truth values to some Boolean variables, we obtain a set of inequalities (denoted by Ψ) as follows.

```
$\Psi$ is empty initially;
for each Boolean variable $b$ which has a value in $pSol$ {
    if $b$ is TRUE, add prCons($b$) to $\Psi$;
    otherwise, add the negation of prCons($b$) to $\Psi$;
}
```

Then we use linear programming to decide the feasibility of Ψ. If the inequalities are found to be infeasible, the algorithm backtracks.

A Boolean variable may be assigned FALSE in a partial solution, yet we cannot give the constraint $exp_1 \neq exp_2$ to a LP package. That is why we have to transform equations into inequalities before the backtracking search procedure is started. Otherwise, suppose there is a Boolean variable b, and $prCons(b)$ is $x = y + 3$. If it happens that, during the search, b becomes false in the partial solution. Then Ψ contains the constraint $x \neq y + 3$, and it is impossible to check the feasibility of Ψ by linear programming.

Example 2. (Cont'd) The search tree is like the following:



There are three branches:

1. In the first branch, `b4` is false. From this assignment and the second Boolean formula `imp(or(married,b2),b4)`, we know that both `married` and `b2` are false. Thus the third Boolean formula `and(b2,not(b1))` cannot be satisfied.
2. In this partial solution, `b4` is true, but `b3` is false. Again, contradiction occurs as a result of Boolean reasoning. From the last Boolean formula `or(b3,married)`, we know that `married` must be true. Then the first formula `imp(married,b1)`

forces `b1` to be true. And the third Boolean formula `and(b2,not(b1))` cannot be satisfied.

3. Both `b3` and `b4` are true. Contradiction occurs, because `b3` stands for `(salary > 10000)`, `b4` stands for `(salary < 8000)`, and it is impossible for `salary` to be less than 8000 and greater than 10000. This kind of inconsistency is detected by a linear programming package.

Non-linear constraints may also be given to BoNuS. But we do not expect that all such problems can be solved. For details about the non-linear case, see [13].

4.4. *Arrays*

To use a tool like BoNuS, we need to transform arrays in the constraints into simple variables. There are two methods for doing this. The first way is to treat syntactically different array expressions as different variables. For example, the constraint $a[i+1] > k-2$ is regarded as $\_a\_i\_plus\_1 > k-2$, where $\_a\_i\_plus\_1$ is a new variable. It is not difficult to see that, when the new constraints are unsatisfiable, so are the original constraints. But the converse is not true.

A more accurate way is as follows. For an array a of size l, we introduce l simple variables: $\$a_0, \$a_1, \ldots, \$a_{l-1}$. An array expression $a[exp]$ will be replaced by such a simple variable, depending on the value of *exp*. Suppose $l = 3$, then a constraint like $a[x+y] > 5$ will be translated to the following:

```
((x+y == 0) && $a0 > 5)) ||
((x+y == 1) && $a1 > 5)) ||
((x+y == 2) && $a2 > 5))
```

This method is more expensive, because it introduces more variables and generates more constraints.

Example 3. In [14], an execution trace of a program is given. But there is a typo in the code, and the path is actually not executable. To see this, we may give the following as input to PAT.

```
  int max, min, sum, i, k, n, a[6];
{   max = a[1];
    min = a[1];
    sum = a[1];
    i = k+1;
  @ i <= n;
  @ max < a[i];
    sum = a[i];
  @ min > a[i];
    sum = sum + a[i];
    i = i+k;
    @ i <= n;
    @ max < a[i];
      sum = a[i];
    @ min > a[i];
      sum = sum + a[i];
      i = i+k;
    @ i <= n;
  }
```

PAT generates the following constraints:

```
    k+1-n <= 0;
    a[1]-a[k+1] < 0;
    a[1]-a[k+1] > 0;
    2k+1-n <= 0;
    a[1]-a[2k+1] < 0;
    a[1]-a[2k+1] > 0;
    3k+1-n <= 0;
```

It is easily seen that they are unsatisfiable. But we also tried BoNuS on the problem. If the first method is used, there are 5 integer variables (i.e., k, n, $a[1]$, $a[k+1]$ and $a[2k+1]$). BoNuS completes the search in less than 0.001 second. For the second method, there are 8 integer variables, and the running time of BoNuS is about 0.07 second. The timings are obtained on a SUN SPARC server with two 400 MHz CPUs and 1024 MB memory,

Example 4. Let us consider a program called SAMPLE ([15], page 65; [16], page 60). An adapted version of it is given in Fig. 3, and the following is one of the paths.

```
   int fa, fb, i, t;
   int a[4], b[4];
 {
     i = 1;      fa = 0;      fb = 0;
   @ i <= 3;   @ a[i] == t;   fa = 1;    i = i+1;
   @ i <= 3;   @ a[i] != t;   i = i+1;
   @ i <= 3;   @ a[i] != t;   i = i+1;
   @ i > 3;    @ fa == 1;
     i = 1;      fb = 1;
   @ i <= 3;   @ b[i] == t;   i = i+1;
   @ i <= 3;   @ b[i] == t;   i = i+1;
   @ i <= 3;   @ b[i] == t;   i = i+1;
   @ i > 3;    @ fb == 1;
 }
```

Given the above input, PAT produces the following constraints:

```
 int fa, fb, i, t;
 int a[4], b[4];
{
 i = 1;
 fa = 0;
 fb = 0;
 while (i <= 3) {
  if (a[i] == t)
    fa = 1;
  i = i+1;
 }
 if (fa == 1) {
  i = 1;
  fb = 1;
  while (i <= 3) {
    if (b[i] != t)
     fb = 0;
    i = i+1;
  }
 }
 if (fb == 1)
  out = 1;
 else out = 0;
}
```

Fig. 3. The program SAMPLE.

```
a[1]-t == 0;  a[2]-t != 0;  a[3]-t != 0;
b[1]-t == 0;  b[2]-t == 0;  b[3]-t == 0.
```

It is easy to decide that the constraints are satisfiable, either by hand or by BoNuS.

We have studied many examples from the literature. In most cases, the feasibility of paths can be decided very quickly. Some experimental results are summarized in Table 1. In the table, t_0 and t_1 denote the time of obtaining path conditions and the time of constraint solving, respectively. We use a SUN SPARC server with two 400 MHz CPUs and 1024 MB memory, and the timings are measured in seconds. Each row (except for the first row) of the table corresponds to a path. The paths are extracted from several different programs:

- *fab_i*: paths of the program in Example 4.
- *mid_i*: paths of the program `middle` [17, 13].
- *grd_i*: paths of the program `grader` [18].
- *ist_i*: paths of the insertion sort program.

Table 1. Experimental results.

Path	feasible?	t_0	t_1
fab_1	Yes	0.00	0.01
fab_1s	No	0.01	1.51
fab_2	Yes	0.00	0.02
fab_2s	No	0.00	5.04
fab_3	Yes	0.01	0.00
fab_3s	No	0.01	0.01
mid_1s	Yes	0.01	0.09
mid_2s	No	0.01	0.05
mid_3s	No	0.00	0.05
mid_4s	No	0.01	0.03
grd_1	Yes	0.02	0.00
grd_2	Yes	0.03	0.10
ist_1	Yes	0.01	0.00
ist_2	Yes	0.02	0.10
ist_3	Yes	0.00	0.00
bst_1s	No	0.00	0.00
bst_2s	Yes	0.08	0.01
bst_3s	Yes	0.08	0.03
bst_4s	Yes	0.00	0.01

- bst_i: paths of the bubble sort program.

If a path name ends with 's', it means that a specification (i.e., the negation of the postcondition) is added to the end of the path. For example, the last row says there is a path of the bubble sort program, which leads to the violation of the postcondition. (We modified the standard program slightly.)

5. Related Work

There has been a large amount of work on test data generation, some of which use constraint-based techniques. For example, a set of tools for test data generation, called Godzilla, was described in [8]. It includes a path analyzer, a constraint generator and a constraint satisfier. Godzilla keeps all constraints in the Disjunctive Normal Form (DNF). The constraint solving method used in Godzilla is incomplete. It may not be able to find a solution even when the constraints are satisfiable.

Similarly, optimization methods like simulated annealing [17] and genetic algorithms [19] have been used to generate test data. They are automatic, and effective in many cases. But they cannot be used to prove that a path is infeasible.

Some heuristic rules for identifying infeasible constraints are given in [20]. Most of the rules are very simple. An example is the following:

> For any two constants k_1 and k_2, if $(k_1 > k_2)$, then the two constraints $(x > k_1)$ and $(x < k_2)$ conflict.

These rules only provide a partial solution to the feasibility problem, even under the restriction that all numeric expressions are linear.

In contrast to the incomplete methods, Constraint Logic Programming (CLP) systems can find a solution if the constraints are satisfiable, and they can also detect the unsatisfiability of constraints. (For more information about CLP, see the survey [21].) Examples of CLP systems include Prolog III, CHIP and CLP($\mathcal{R}$). Some of them are quite expressive and efficient. But it is not easy to use them directly for path feasibility analysis. Most of them are based on Horn clauses. Yet the constraints generated from program paths can be non-Horn or even non-clausal. (An example of such formulas, which is quite complicated, is given at the end of [22].) More recent constraint programming systems like ILOG Solver (`http://www.ilog.fr`) are not restricted to the Horn logic.

Many CLP systems support only a few data types (such as finite domain integers and real numbers). For example, the test data generation tool INKA [16] generates constraints and then solves them using CLP(FD). The latter is a finite-domain constraint solver and it is an extension of Prolog. All variables are integers that can take values from some finite domains, e.g., $x \in [1, 9]$. The system CLP(R) [23] can deal with real numbers, but it has difficulty with linear inequalities. Eliminating variables from the inequalities is very expensive.

In addition to the above methods, one may also use general-purpose theorem provers or checkers. For example, the resolution-style theorem prover KITP has been suggested as the main tool to determine the feasibility of test specifications [24]. The user tries to prove a conjecture using standard axioms of arithmetic and conditional rewrite rules. This approach is very general, but it can hardly be automated.

In most cases, when the numeric expressions are linear, a better alternative is to use decision procedures for Presburger arithmetic to detect non-executable paths. Presburger formulas are first-order logical formulas defined on the integer domain, with the restriction that the arithmetic operations on integer variables can only be addition and subtraction. Non-linear numeric expressions are not allowed. But the formulas may have universal and existential quantifiers ($\forall$, $\exists$). Presburger arithmetic is decidable, but the decision procedures suffer from very high worst-case complexity [25]. For the purpose of feasibility analysis, we think that it is too expressive. If the constraints are to be represented as logical formulas, only existential quantifiers are needed. The path exploration tool PET [26] uses the Presburger arithmetic decision procedure built in the prover HOL. In the input language of PET, the variables can only be integers.

Besides path-oriented testing, constraint solving techniques can be applied in other ways to increase the reliability of software. For instance, in specification-based testing, we need to decide the satisfiability of constraints [27]. In [13], it is shown that tools like BoNuS can be used to check potential inconsistency and incompleteness in state-based requirement specifications.

6. Concluding Remarks

A major difficulty with path-oriented testing methods is that many paths turn out to be non-executable or infeasible. This also occurs frequently in the assertion-based analysis approach, where a postcondition is negated. (Unless the program contains many bugs, the negated postcondition is usually not satisfiable.)

To determine the feasibility of program paths, we may generate a set of constraints and then decide their satisfiability. This can be done by hand or with the help of various tools (such as theorem provers, decision procedures and constraint solvers). We believe that the constraint solving approach is better in terms of efficiency and degree of automation. However, many existing constraint solvers are not so easy to use for the purpose of program testing. Our experiences indicate that the constraints are generally not too difficult to solve, but they may be complicated syntactically. There can be variables of different types and many Boolean operators.

In this paper, we describe constraint-based tools for determining the feasibility of paths. The tool PAT extracts path conditions, while BoNuS is used to solve them. They can be combined to generate test data and to detect infeasible paths. The tools are highly automatic, and they allow the user to specify constraints and paths in a very natural way. Common data types such as integers, Booleans, and arrays are accepted. PAT can also be used as a front-end to constraint solvers like BoNuS, since a "path" can contain logical expressions only (without assignment statements). The tools may also be combined with other path selection methods and tools (e.g., that of [4]), to generate executable test cases.

One problem with our approach is that LP packages like `lp_solve` suffer from rounding errors when dealing with real numbers. This is not too serious, since in many cases, there is not much numerical computation, and the constraints are either unsatisfiable or easily satisfiable. The main drawback of our approach is the complexity of the underlying algorithms. We do not expect that it works on large programs. However, bad theoretical results do not necessarily imply that a particular problem instance cannot be solved quickly. Moreover, the performance of computer hardware has been improved rapidly. In the future, we will make improvements to the search algorithm and find better ways to deal with arrays. We also plan to study other data types such as pointers and character strings. We believe that our tools are useful in the evaluation of algorithms and in the unit testing phase of software development.

Acknowledgements

The authors would like to thank the anonymous reviewers for careful reading of an earlier draft and for their suggestions.

References

1. D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs", *Proc. 8th Int. Conf. on Software Engineering* (1985), pp. 259–266.

2. E. J. Weyuker, "An empirical study of the complexity of data flow testing", *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada (1988), pp. 188–195.
3. B. Korel and A. M. Al-Yami, "Assertion-oriented automated test data generation", *Proc. 18th Int. Conf. on Software Engineering* (1996), pp. 71–80.
4. A. Bertolino and M. Marré, "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. Softw. Eng.* **SE-20** (1994) 885–899.
5. E. Weyuker, "Translatability and decidability questions for restricted classes of program schemas", *SIAM J. on Computing* **8** (1979) 587–598.
6. L. J. White and E. I. Cohen, "A domain strategy for computer program testing", *IEEE Trans. Softw. Eng.* **SE-6** (1980) 247–257.
7. L. A. Clarke and D. J. Richardson, "Symbolic evaluation methods — Implementations and applications", in *Computer Program Testing*, eds. B. Chandrasekaran and S. Radicchi (North-Holland, Amsterdam, 1981), pp. 65–102.
8. R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation", *IEEE Trans. Softw. Eng.* **SE-17** (1991) 900–910.
9. A. K. Mackworth, "Constraint satisfaction," in *Encyclopedia of Artificial Intelligence*, (ed.) S. C. Shapiro, Vol. 1 (John Wiley, New York, 1990), pp. 205–211.
10. P. R. Thie, *An Introduction to Linear Programming and Game Theory* (John Wiley, New York, 1979).
11. P. D. Coward, "Symbolic execution and testing", *Information and Software Technology* **33** (1991) 53–64.
12. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W. H. Freeman, San Francisco, 1979).
13. J. Zhang, "Specification analysis and test data generation by solving Boolean combinations of numeric constraints", *Proc. First Asia-Pacific Conference on Quality Software*, Hong Kong (Oct. 2000), pp. 267–274.
14. B. Korel and S. Yalamanchili, "Forward computation of dynamic program slices", *Proc. Int'l Symp. on Software Testing and Analysis* (*ISSTA'94*), Seattle, Washington, USA (1994), pp. 66–79.
15. R. Ferguson and B. Korel, "The chaining approach for software test data generation", *ACM Trans. on Softw. Eng. and Methodology* **5** (1996) 63–86.
16. A. Gotlieb, B. Botella and M. Rueher, "Automatic test data generation using constraint solving techniques", *Proc. Int'l Symp. on Software Testing and Analysis* (*ISSTA'98*), Clearwater Beach, Florida, USA (1998), pp. 53–62.
17. N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing", *Proc. Int'l Symp. on Software Testing and Analysis* (1998), pp. 73–81.
18. B. Korel and J. Laski, "STAD — A system for testing and debugging: User perspective", *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada (1988), pp. 13–20.
19. R. P. Pargas, M. J. Harrold and R. R. Peck, "Test-data generation using genetic algorithms", *Software Testing, Verification and Reliability* (1999).
20. A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability* **7** (1997) 165–192.
21. J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *J. of Logic Programming* **19/20** (1994) 503–581.
22. A. Goldberg, T. C. Wang and D. Zimmerman, "Applications of feasible path analysis to program testing", *Proc. Int'l Symp. on Softw. Testing and Analysis* (*ISSTA'94*), Seattle, Washington, USA (1994), pp. 80–94.
23. N. C. Heintze, J. Jaffar, S. Michaylov, P. J. Stuckey and R. H. C. Yap, "The CLP($\mathcal{R}$)

programmer's manual", Version 1.2, Sept. 1992.

24. R. Jasper, M. Brennan, K. Williamson, B. Currier and D. Zimmerman, "Test data generation and feasible path analysis", *Proc. Int'l Symp. on Software Testing and Analysis* (*ISSTA'94*) Seattle, Washington, USA (1994) pp. 95–107.
25. D. Oppen, A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic, *J. of Computer and System Sciences* **16** (1978) 323–332.
26. E. L. Gunter and D. Peled, Path exploration tool, *Proc. of the 5th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'99*), Amsterdam, The Netherlands (1999), Lecture Notes in Computer Science, Vol. 1579, pp. 405–419.
27. A. J. Offutt and S. Liu, "Generating test data from SOFL specifications", *J. of Systems and Software* **49** (1999) 49–62.